Lab Assignment #5

Guideline

This lab can be done with a partner.

Objectives

- To implement a stack calculator.
- To get more familiar with block RAMs on an FPGA and understand memory interfacing
- To understand how to model buses in Verilog.

Description

In this lab, you will write Verilog code to implement a stack calculator using a memory module (block RAM on the FPGA) and the board I/O.

A stack calculator stores its operands in a stack structure and performs operations (e.g. addition, subtraction, etc) on the top two values of the stack. The operands are popped off the stack and the result is pushed back on the top of the stack, so the stack is one element less than it was before the operation. The output of the calculator is always the value at the top of the stack. The stack may contain more than two operands at any time, but operations are only performed on the top two values. This is similar to the Reverse Polish Notation (RPN) used by old TI calculators.

You will implement a simple stack calculator using Xilinx BlockRAM as the storage for the stack. We will provide you with code to implement/model a memory using the BlockRAM. The memory supplied is byte-addressable and 8 bits (1 byte) wide. Please check the synthesis report to make sure that Xilinx Vivado is synthesizing your design using a BlockRAM and not distributed LUT-RAM.

We will also provide you the code for top block which integrates the controller, memory along with the data bus. You will need to modify the supplied code to implement the bus interface to memory and the controller. This will involve using tri-state buffers. Through tri-state buffers, we will be able to ensure that only one driver drives the data bus at a time.

You will also need to implement a master controller for the calculator. This controller contains three registers, a stack pointer (SPR), a display address register (DAR), and a display value register (DVR). The SPR will contain the address of the next free address past the top of the stack. The DAR will hold the address of the data that should be displayed on the output. Whenever the SPR is updated, the DAR should be updated to SPR plus 1. The DVR contains the value that should be displayed on the output. The content of the DVR is the value stored at the memory location pointed to by the DAR. This should be updated every time the DAR changes by reading from the memory location contained in the DAR. The memory will be 128 bytes total giving a 7-bit address. So the SPR and DAR will both be 7 bits wide and the DVR will be 8 bits wide. The master controller will be responsible for taking in the inputs, updating its registers accordingly, as well as performing the operations of the calculator and displaying the outputs. Before use, a reset/clear operation should be used to initialize the calculator. The SPR should be initialized to 0x7F, the DAR to 0x00, and the DVR to 0x00 (don't read from memory this time). As data is pushed on to the stack, the SPR will decrement. In other words, the stack will grow towards decreasing addresses.

The calculator will use all of the inputs/outputs on the Xilinx board. The seven segment displays will show the contents of the DVR (only two of them will be used because the data size is 8 bits). Values will be entered, 8 bits at a time, using the switches on the board (SW0 maps to the LSB). LED[6:0] show the contents of the DAR bits 6 down to 0 and LED[7] will map to an 'EMPTY' flag. If the stack is empty (the SPR contains 0x7F), the EMPTY flag (LED[7]) will be set to '1'.

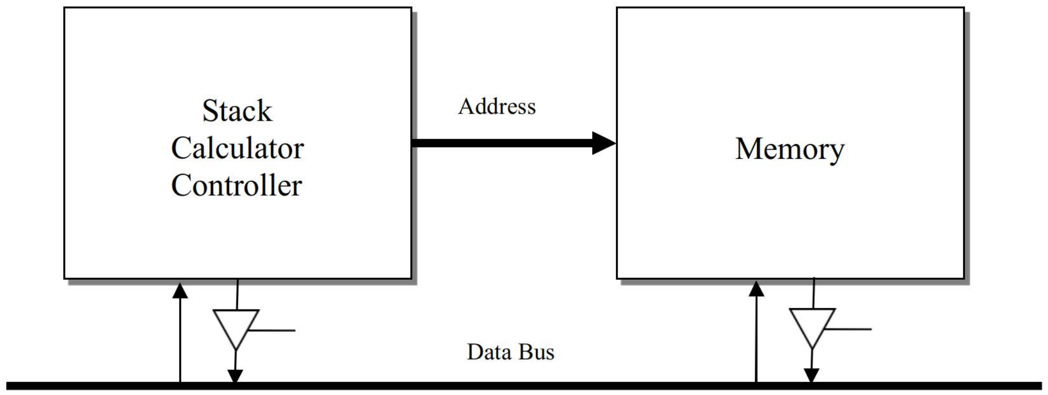
The buttons will provide the operational inputs to the controller. Each button will implement a function as defined in the table below:

Mode	**BTN3 BTN2**	**BTN1**	**BTN0**
Push/Pop	0 0	Delete/Pop	Enter/Push
Add/Subtract	0 1	Subtract	Add
Clear/Top	1 0	Clear/RST	Top
Dec/Inc	1 1	Dec Addr	Inc Addr

- **Enter/Push**: Reads the value from switches on the board and pushes it on the top of the stack. To do this, keep BTN3 and BTN2 at 0 (ie. unpressed) and press BTN0
- **Delete/Pop**: Pops and discards the 8-bit value on the top of the stack. To do this, keep BTN3 and BTN2 at 0 (ie. unpressed) and press BTN1
- **Add**: Pops the top two 8-bit values on the stack, adds them, and pushes the 8-bit result on the top of the stack, discarding the carry bit. To do this, keep BTN3 at 0 (unpressed) and BTN2 at 1 (pressed) and press BTN0
- **Subtract**: Pops the top two 8-bit values on the stack, subtracts them, and pushes the 8-bit result on the top of the stack, discarding the borrow bit (High Addr minus Low Addr). To do this, keep BTN3 at 0 (unpressed) and BTN2 at 1 (pressed) and press BTN1
- **Clear/RST**: Resets the SPR to 0x7F, the DAR to 0x00, and the DVR to 0x00. The stack should be empty now (EMPTY flag should be set to 1). To do this, keep BTN3 at 1 (pressed) and BTN2 at 0 (unpressed) and pressBTN1
- **Top**: Sets the DAR to the top of the stack (SPR+1; will cause the DVR to update). To do this, keep BTN3 at 1 (pressed) and BTN2 at 0 (unpressed) and press BTN0
- **Dec Addr**: Decrements the DAR by 1. To do this, keep BTN3 1 (pressed) and BTN2 at 1 (pressed) and press BTN1
- **Inc Addr**: Increments the DAR by 1. To do this, keep BTN3 1 (pressed) and BTN2 at 1 (pressed)

and press BTN0 The following figure shows the block diagram of your design:

Top-Level Design



Useful Information

1. Note that the overflow, underflow, and pushing both BTN0 and BTN1 at the same time are not considered in this lab. In general you can assume the calculator will be used as described, i.e. you do not have to worry about the error conditions like POPing without having pushed anything, decrementing DAR beyond the lowest address. Keep it simple.
2. All data on the stack should be considered unsigned.
3. Also note that the INC and DEC commands affect the DAR and DVR. They don't modify SPR. INC and DEC are just to be able to see the contents of various locations on the stack. Similarly, the POP/PUSH/ADD/SUBTRACT will use the SPR (however, they will update the DAR and DVR as well).
4. Simulation is NOT a requirement to get full credit if your design works perfectly on the board. If it does not work on the board, please have simulation ready for partial credit.
5. The simplest way to approach the design of the controller is to use a large state machine. The first state will be state will waits for inputs from the user. You will jump from this state to others depending on the inputs.
6. The memory works on 50MHz, but has single cycle latency. This means that when reading from the memory, if you make WE 0 in one clock cycle (in other words, in one state of the controller), you should read data from the data bus in the next clock cycle (in other words, in the next state of the controller). Similarly, when writing to the memory, you should make WE 1 in one clock cycle and wait for one clock cycle to let the memory write the data.
7. The controller can use as many cycles (states) as it wishes to perform the tasks (like POP, PUSH, ADD, etc). Since the clock frequency is 50MHz, even if the controller takes 10 cycles (say) to perform an operation, the user won’t be able to see the lag with the naked eye.
8. If you need, you can modify the ports of the modules given to you. But we would want you to not change the memory module at all.

Submission Details

Submit the following files through Canvas. No hard copy submission is required.

- Typed Verilog Code (.v files)
- [filename].bit file

Checkout Details

Demonstrate the calculator working on the board to the TA during checkout.

Example

Suppose you want to add 0x92 (binary form: 10010010) and 0x25 (binary form: 00100101). You should first push these two numbers on the top of the stack (the stack at this time can contain other numbers). Perform the following sequence to enter 0x92 and then 0x25 into the stack:

0. Reset the calculator, hold BTN3 and push BTN1. At this point, LED[7:0]=1_0000000 and 7-seg are 00
1. Set the switches (SW7 down to SW0) to 10010010.
2. Push BTN0. At this point, LED[7:0] are 0_1111111 and the 7-segs are 92.
3. Set the switches (SW7 down to SW0) to 00100101.
4. Push BTN0. At this point, LED[7:0] are 0_1111110 and the 7-segs are 25.
5. Hold BTN2 and push BTN0. At this point, LED[7:0] are 0_1111111 and the 7-segs are B7.
6. After steps 2 or 4, you can pop (delete) the numbers you have entered by pressing BTN1.

Starter Code

Top Module

```
module top(clk, btns, swtchs, leds, segs, an);
  input clk;
  input[3:0] btns;
  input[7:0] swtchs;
  output[7:0] leds;
  output[6:0] segs;
  output[3:0] an;

  //might need to change some of these from wires to regs
  wire cs;
  wire we;
  wire[6:0] addr;
  wire[7:0] data_out_mem;
  wire[7:0] data_out_ctrl;
  wire[7:0] data_bus;

  //CHANGE THESE TWO LINES
  assign data_bus = 1; // 1st driver of the data bus -- tri state switches
                       // function of we and data_out_ctrl

  assign data_bus = 1; // 2nd driver of the data bus -- tri state switches
                       // function of we and data_out_mem


  controller ctrl(clk, cs, we, addr, data_bus, data_out_ctrl,
      btns, swtchs, leds, segs, an);

  memory mem(clk, cs, we, addr, data_bus, data_out_mem);

  //add any other functions you need
  //(e.g. debouncing, multiplexing, clock-division, etc)

endmodule
```

Controller

```
module controller(clk, cs, we, address, data_in, data_out, btns, swtchs, leds, segs,an);
  input clk;
  output cs;
  output we;
  output[6:0] address;
  input[7:0] data_in;
  output[7:0] data_out;
  input[3:0] btns;
  input[7:0] swtchs;
  output[7:0] leds;
  output[6:0] segs;
  output[3:0] an;


  //WRITE THE FUNCTION OF THE CONTROLLER

endmodule
```

Memory

```
module memory(clock, cs, we, address, data_in, data_out);
  input clock;
  input cs;
  input we;
  input[6:0] address;
  input[7:0] data_in;
  output[7:0] data_out;

  reg[7:0] data_out;

  reg[7:0] RAM[0:127];

  always @ (negedge clock)
  begin
    if((we == 1) && (cs == 1))
      RAM[address] <= data_in[7:0];

    data_out <= RAM[address];
  end
endmodule
```